

C/C++ Contributing Editors

The New C: bool, Advice to C and C++ Programmers

Randy Meyers

True or False: The type **bool** behaves identically in C99 and C++. True or False: Your C89 **bool** hackery will survive unscathed under C99. Read and find out.

There is nothing new under the sun. As I wrote in my first installment of “The New C” [\[1\]](#), the C99 Standard contains almost nothing new. Except for two or three features, C99 standardized extensions from existing compilers. So, even if you are using a C compiler that has not been updated in several years, it might contain little pieces of C99. For example, variable length arrays [\[2, 3, 4, 5\]](#) came from the Cray C compiler, and a similar feature is in GNU C. Compound literals [\[6\]](#) and designated initializers [\[7\]](#) came from the Plan 9 C compiler. Lots of compilers have supported **long long** [\[8\]](#) for a long, long time. Restricted pointers [\[9\]](#) have been in compilers for high performance systems for years. The good news for programmers is that if you are using an old compiler that supports an extension that is now part of C99, you need not fear the portability consequences of using that extension quite so much.

There is another aspect of C99 that I find profound (but I am easily awed). There are at least two parts of C99 that any programmer can add to an older compiler simply by creating a header file. The first is the generalized integer support [\[10\]](#) in the C99 `<stdint.h>` and `<inttypes.h>` headers. The second is subject of this column, the **bool** type.

The **bool** type came from C++. When I wrote that C99 standardized extensions from existing compilers, I was careful not to over qualify my statement by writing “existing C compilers.” C++ was the source of several features in C99 including mixed declarations and code and new lifetime and initialization rules [\[7\]](#). The C99 committee felt strongly that the **bool** type in C99 should be compatible with C++ and should serve to broaden the intersection of the two languages. However, the C99 committee also had to introduce **bool** into C99 in such a way as to limit its impact on the existing C code base: moving even large programs from C90 to C99 should be nearly effortless. The C99 committee solved this problem by in effect making the new **bool** type conditional on whether a new header `<stdbool.h>` is included. The `<stdbool.h>` header may be included by C++, where it does nothing. This allows the same source code to be compiled either by C or C++, a particularly important feature when writing header files. Placing **bool** support in `<stdbool.h>` also allows programmers without access to C99 to create their own `<stdbool.h>` and get an approximation to the C99 and C++ feature.

In this column, I am going to discuss the **bool** type in C99 and describe how to write your own `<stdbool.h>` if you do not have access to C99.

Compatibility with Old Code

Strictly speaking, there is no functional need for the **bool** type. Any program that needs a flag can use an **int** instead. However, even with the crummy variable name,

the declaration **bool x**; communicates a great deal of information to the programmer about the purpose of variable **x** and how variable **x** is likely to be used in expressions. Likewise, the expression **y = true**; communicates information lacking in the computationally equivalent **y = 1**;. Not surprisingly, many C programmers have their own homegrown version of **bool** to increase the expressiveness of their programs.

C99 followed C++'s lead and added the type **bool** and the two boolean constants **true** and **false**. In C++, **bool**, **true**, and **false** are keywords in the language. This is a reasonable approach given the pervasive use of overloading involving **bool** in C++ and its library. However, it does mean that C programs using homegrown **bool** being ported to C++ frequently encounter compilation errors due to the C++ keywords. Worse, if the homegrown **bool** uses macros, the C program may hijack the C++ type system causing overloading and the C++ library to malfunction.

C99 lacks user overloading, and **bool** is not used throughout the C library. Thus, C99 has few problems coexisting with a homegrown **bool**. To avoid invalidating homegrown versions of **bool**, C99 declares **bool**, **true**, and **false** as macros in the header `<stdbool.h>`. If `<stdbool.h>` is not included, the program can ignore the new boolean type in C99 and can use the names **bool**, **true**, and **false** in any way it pleases. Even when `<stdbool.h>` is included, there are some provisions for accommodating a homegrown **bool** (discussed below).

C99 does add one new keyword: **_Bool**. This is the boolean type built into the C99 compiler. The **bool** macro in `<stdbool.h>` merely expands into **_Bool**. Note that since 1989 the C Standard has reserved all identifiers beginning with an underscore followed by either another underscore or a capital letter. So, the **_Bool** keyword should not interfere with existing code. Note also that the intended way to use the boolean type in C99 is to include `<stdbool.h>` and use **bool**. Not only is **bool** more tasteful and easier to type, it is also compatible with C++. Never use the **_Bool** keyword directly.

The Boolean Type

Unless I specifically write otherwise, C and C++ have the same semantics for **bool**.

The **bool** type only holds the values **true** and **false**. While this requirement only necessitates a single bit of storage, **sizeof** still needs to return an integral value for **sizeof(bool)**, and you can take the address of a **bool** using the **&** operator. Thus, **bool** must take up at least a byte of storage, but both C and C++ allow the compiler to use more than one byte. For example, some compiler might make **sizeof(bool)** equal **sizeof(int)**.

The **bool** type is considered to be a member of the integer family of types. Thus, an expression of type **bool** can be used wherever an expression of type **int** can be used. (C++ makes an exception for the **++** and **-** operators). When a numeric value is needed for a value of type **bool**, the **bool** automatically promotes to **int**. If the **bool** value is **true**, its value becomes **1**. If the **bool** value is **false**, its value becomes **0**.

Arithmetic types and pointer types may be converted to **bool**. However, this is not the normal conversion to an integer type. A conversion to **bool** converts the value to **true** or **false** the same way that the **if** statement determines whether its controlling expression is **true** or **false**: it implicitly compares the value against zero. Thus,

```
void ex1(char c, int i, double d, char *p)
{
    bool b1 = c;
    bool b2 = i;
    bool b3 = d;
    bool b4 = p;
}
```

is compiled exactly as if it had been written:

```
void ex1(char c, int i, double d, char *p)
{
    bool b1 = c != '\0';
    bool b2 = i != 0;
    bool b3 = d != 0.0;
    bool b4 = p != NULL;
}
```

The conversion to **bool** works the same regardless of whether it is an explicit or an implicit conversion. Thus, all legitimate ways of getting a value into a **bool** result in the **bool** being either **true** or **false**, **1** or **0**. If the **bool** is uninitialized or has received a value through some invalid method (assignment through the wrong member of a union or via a mistyped pointer), the program has undefined behavior if it tries to use the value of the **bool**. Whether such a “bad” **bool** is either **true** or **false** is undefined.

The two boolean constants are **true** and **false**, which have the numeric values **1** and **0**, respectively. In C99, these constants have type **int**. In C++, they have type **bool**. A very minor incompatibility between C99 and C++ is that **sizeof(bool)** might not equal **sizeof(true)** in C99. In C99, **true** and **false** are just macros that expand to **1** and **0**, respectively.

true and **false** may be used in the controlling expressions of preprocessor **#if** directives. While this is part of both C99 and C++, I’ve encountered a few C++ compilers that issue errors about this. Also beware: a standard feature of the preprocessor is that any identifiers used in the controlling expression of **#if** that are not defined to be macros are replaced with the constant **0**. Thus, a preprocessor may consider **true** to be **0** (**false**) if **<stdbool.h>** is not included when compiling C99. This problem can also occur when compiling C++ if the preprocessor has not been updated to understand **true** and **false** for C++.

Bitfield members may have type **bool**. A **bool** bitfield never sign extends. Thus, if you store **1** in a one-bit **bool** bitfield, the value of that bitfield will be **1** (a clean **true**) and never **-1**. For this reason, C99 considers **bool** to be an unsigned type. C++ requires this semantics for **bool** bitfields, but does not explicitly say **bools** are unsigned.

The behavior of **bools** in expressions follows from the rules above. Interesting cases are the **++** and **--** operators. **++** adds one to its operand. If the operand is a **bool**, its value is converted to an **int** in order to perform the operation, and then the result is converted back to a **bool** to be stored. Thus, if you increment a **bool** whose value is **false**, you add **1** to zero getting one, which converts back to **true**. If you increment a **bool** whose value is **true**, you add **1** to one getting two, which converts back to **true**. Thus, incrementing a **bool** always sets its value to **true**. Likewise, decrementing a **bool** sets the **bool** to its logically complemented value (think it through). C++ allows

incrementing a **bool** (but marks the feature as deprecated, which means it might be removed in a future standard). C++ forbids decrementing a **bool**. C99 allows both.

<stdbool.h> Coexistence

In addition to defining macros for **bool**, **true**, and **false**, <stdbool.h> defines the macro **__bool_true_false_are_defined** to be **1**. This warns a homegrown version of **bool** that <stdbool.h> has been included and has defined macros for **bool**, **true**, and **false**.

The homegrown version of **bool** might check **__bool_true_false_are_defined** and not define its own versions of **bool**, **true**, and **false**. This might be useful if the homegrown **bool** is in a project-wide header file that is included everywhere, but a few modules have been converted to use the C99 version of **bool**.

The homegrown version of **bool** might also check **__bool_true_false_are_defined** in order to undefine the macros from <stdbool.h>. Normally, it is undefined behavior to undefine a macro from a standard header. However, C99 permits **bool**, **true**, and **false** from <stdbool.h> to be undefined and possibly redefined to something else. This is considered a short-term concession to coexistence with homegrown versions of **bool**. The feature is marked obsolescent, and the next version of C99 may withdraw this permission.

Roll Your Own <stdbool.h>

Although you cannot fully add **bool** to a pre-C99 compiler merely by writing a header file, you can come close. The only thing lacking is that conversions to your own **bool** will not have the proper semantics of testing the value being converted against zero. If you write such comparisons explicitly, you can write your own version of <stdbool.h> and program portably between C90, C99, and C++.

[Listing 1](#) shows my version of <stdbool.h>. Note that no macros are defined if the header is included from C++ in order to not interfere with C++'s built-in **bool**. If the file is compiled by C99, **bool** is defined to be **_Bool** as required by the C99 Standard. If included from pre-C99, **bool** is defined to be an unsigned type suitable for use as a bitfield. By making the type unsigned, bitfields of length **1** will not sign extend and thus meet the requirements of C99 and C++. Not all compilers permit unsigned **char** bitfields, but it is a popular extension. If your compiler complains, define **bool** to be unsigned **int**.

[Listing 2](#) is a short test program that checks our version of <stdbool.h>. The program can be run under C90, C99, or C++. Note that **stdbool.h** is included using the quoted form of the directive. This allows the compiler to find our **stdbool.h** even though it is not located with the system **includes**. If an official C99 version becomes available, merely deleting our own file will cause the compiler to find the system version since the quoted **#include** searches the system area if it cannot find the header in the user area. Some compilers have options to allow additional directories to be searched when the <> form of **#include** is used. Under such a compiler, **#include <stdbool.h>** would work.

C99 and C++ should pass the test in [Listing 2](#) without any warnings from the test program run. A few C++ compilers will issue compile-time warnings about the implicit conversions to **bool** and perhaps using **bool** with a relational operator. These operations are valid C++, but sometimes misused, and so the compilers are trying to

be helpful. A pre-C99 compiler should only get one warning that the conversion to **bool** does not compare against zero.

One popular C++ compiler issues errors and fails to compile the **#if** directives that test whether **true** and **false** can be used in the preprocessor. If you encounter this compiler bug, feel free to delete or comment out the tests.

Conclusion

C99 makes programs more readable and increases the overlap between C and C++ by adding the **bool** type. Programs using **bool** can be compiled under either C99 or C++ by including `<stdbool.h>`, and by defining **bool**, **true**, and **false** in `<stdbool.h>`. C99 avoids keyword problems that might trouble old C code. Pre-C99 programmers or even C++ programmers writing source to be shared with C can write their own version of `<stdbool.h>` without waiting for a C99 implementation to be available.

References

- [1] Randy Meyers. "The New C," *C/C++ Users Journal*, October 2000.
- [2] Randy Meyers. "The New C: Why Variable Length Arrays," *C/C++ Users Journal*, October 2001.
- [3] Randy Meyers. "The New C: Variable Length Arrays, Part 2," *C/C++ Users Journal*, December 2001.
- [4] Randy Meyers. "The New C: Variable Length Arrays, Part 3: Pointers and Parameters," *C/C++ Users Journal*, January 2002.
- [5] Randy Meyers. "The New C: Variable Length Arrays, Part 4: VLA **typedefs** and Flexible Array Members," *C/C++ Users Journal*, March 2002.
- [6] Randy Meyers. "The New C: Compound Literals," *C/C++ Users Journal*, June 2001.
- [7] Randy Meyers. "The New C: Declarations and Initializations," *C/C++ Users Journal*, April 2001.
- [8] Randy Meyers. "The New C: Integers in C99, Part 1," *C/C++ Users Journal*, December 2000.
- [9] Randy Meyers. "The New C: Restricted Pointers," *C/C++ Users Journal*, November 2000.
- [10] Randy Meyers. "The New C: Integers, Part 3," *C/C++ Users Journal*, February 2001.

***Randy Meyers** is a consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at rmeyers@ix.netcom.com.*